Theoretical Computer Science 410 (2009) 406–416

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On small, reduced, and fast universal accepting networks of splicing processors

Remco Loos [a,*], Florin Manea [b], Victor Mitrana [a,b]

[a] Research Group on Mathematical Linguistics, Rovira i Virgili University, Pça Imperial Tàrraco 1, 43005 Tarragona, Spain

[b] Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei 14, 70109, Bucharest, Romania

ARTICLE INFO

Keywords:
Molecular computation
Splicing
Networks of processors

ABSTRACT

In this paper, we show that accepting networks of splicing processors (ANSPs) of size 2 are computationally complete. Since, by definition, an ANSP needs at least two nodes to perform non-trivial computations, this completely settles the question of designing complete ANSPs of minimal size. Also, we derive from this result the fact that all the languages in **PSPACE** can be accepted by ANSPs of size 2, having polynomial length complexity (the ANSP complexity measure for the space used in a computation). However, the construction that we propose, although efficient from the descriptional complexity and space complexity points of view, does not seem to have good properties from the time complexity point of view. In this respect, we prove that ANSPs of size three can decide all languages in **NP** in polynomial time. The previous lower bound on size for both completeness and efficient acceptance of **NP**-languages was seven. We also consider ANSPs with restricted features, proving the following normal forms: for any ANSP there exists an equivalent ANSP without input filters, and one without output filters. Finally, we show how to construct a small universal ANSP and make several considerations on the computational efficiency of universal ANSPs.

1. Introduction

Accepting networks of splicing processors (ANSPs for short) [9,10] are a variant of accepting networks of evolutionary processors, a well-studied model of bio-inspired computing [11–13]. The origin of networks of evolutionary processors (NEP for short) is a basic architecture for parallel and distributed symbolic processing, related to the Connection Machine [7] as well as the Logic Flow paradigm [2], which consists of several processors, each of them being placed in a node of a virtual complete graph, which are able to handle data associated with the respective node. Each node processor acts on the local data in accordance with some predefined rules, and then local data becomes a mobile agent which can navigate in the network following a given protocol. Only such data can be communicated which can pass a filtering process. This filtering process may require to satisfy some conditions imposed by the sending processor, by the receiving processor or by both of them. All the nodes send their data simultaneously and the receiving nodes also simultaneously handle all the arriving messages, according to some strategies, see, e.g., [3,7].

In a series of papers (see [16] for a survey) it was considered that each node may be viewed as a cell having genetic information encoded in DNA sequences which may evolve by local evolutionary events, that is point mutations. Each node is specialized just for one of these evolutionary operations. Furthermore, the data in each node is organized in the form of multisets of words (each word appears in an arbitrarily large number of copies), and all the copies are processed in

* Corresponding address: EMBL-European Bioinformatics Institute, Wellcome Trust Genome Campus, CB10 1SD Hinxton, Cambridge, UK.
E-mail addresses: remcogerard.loos@urv.cat, remco.loos@ebi.ac.uk (R. Loos), flmanea@gmail.com (F. Manea), mitrana@fmi.unibuc.ro (V. Mitrana).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.09.048

parallel such that all the possible events that can take place do actually take place. Obviously, the computational process just described is not exactly an evolutionary process in the Darwinian sense. But the rewriting operations we have considered might be interpreted as mutations and the filtering process might be viewed as a selection process. Recombination is missing but it was asserted that evolutionary and functional relationships between genes can be captured by taking only local mutations into consideration [23].

In the case of accepting networks of splicing processors (ANSPs for short), the point mutations associated with each node are replaced by the missing operation mentioned above, that of splicing. This computing model is similar to some extent to the test tube distributed systems based on splicing introduced in [1] and further explored in [19]. However, there are several differences: first, the model proposed in [1] is a language generating mechanism while ours is an accepting one; second, we use a single splicing step, while every splicing step in [1] is actually an infinite process consisting of iterated splicing steps; third, each splicing step in our model is reflexive; fourth, the filters of our model are based on random context conditions while those considered in [1] are based on membership conditions; fifth, at every splicing step a set of auxiliary words, always the same and proper to every node, is available for splicing. Along the same lines, we want to stress the differences between this model and the time-varying distributed H systems, a generative model introduced in [21] and further studied in [14,20,22]. The computing strategy of such a system is that the passing of words from a set of rules to another one is specified by a cycle. Thus only those words that are obtained at one splicing step by using a set of rules are passed in a circular way to the next set of rules. This means that words which cannot be spliced at some step disappear from the computation while words produced at different splicing steps cannot be spliced together. Now, the differences between time-varying distributed H systems and ANSPs are evident: each node of an ANSP has a set of auxiliary words, words obtained at different splicing steps in different nodes can be spliced together, the communication of words is not done in a circular way, since identical copies of the same word are sent out to all the nodes, the control of communication is accomplished by filters.

A restricted version of ANSPs (in which the splicing operations were performed if and only if exactly one of the two strings spliced was an auxiliary string) was introduced in [9], where it was shown that this computing model is computationally complete. Also, the complexity class **NP** was proved to correspond to the class of languages accepted by restricted ANSPs in polynomial time and **PSPACE** to the class accepted by restricted ANSPs with at most polynomial length of the stings used in the derivation. Finally, a linear time solution for SAT was presented.

In [10] it was proved that ANSPs (unrestricted, this time) of constant size accept (decide) all recursively enumerable (recursive) languages, and can solve all problems in **NP** in polynomial time; also an universality result for ANSPs was proposed. In both cases the number of nodes needed was 7. Here we show that computational completeness can be achieved by ANSPs of 2 nodes. Moreover, in [10] the constant size was achieved using an encoding in a 2-letter alphabet. Here we present a direct construction for any language accepted by a deterministic Turing machine. Also, we show the following normal form results: for any ANSP there exists an equivalent ANSP without output filters, and one without input filters. These results suggest that further research in reduced ANSPs with less features might be fruitful.

For the computational complexity result, we show that ANSPs of size 2 accept all the languages in **PSPACE**, and, further, we prove by a more involved construction that ANSPs of size 3 can simulate the computations of a non-deterministic Turing machine in parallel. We then use this fact to show that ANSPs of size 3 can decide all languages in **NP**. Since, by its definition, ANSPs need at least two nodes to accept any non-trivial language, these results go a long way in settling this issue, leaving, however, one open problem: the efficient simulation of non-deterministic Turing machines by ANSPs with two nodes.

Finally, we consider universal ANSPs. We show how to construct a small universal ANSP and make several considerations on the computational efficiency of universal ANSPs.

2. Basic definitions and notation

We assume the reader's familiarity with the basic concepts in complexity classes and formal language theory. The reader may refer to [8,18] for definitions.

For any set A, $|A|$ denotes the cardinality of A and for a word w, $|w|$ denotes the length of w. The smallest W such that $w \in W^*$ is denoted by $alph(w)$ and the empty word is denoted by λ.

A nondeterministic Turing machine is a construct $M = (Q, V, U, \delta, q_0, B, F)$, where Q is a finite set of states, V is the input alphabet, U is the tape alphabet, $V \subset U$, q_0 is the initial state, $B \in U \setminus V$ is the "blank" symbol, $F \subseteq Q$ is the set of final states, and δ is the partial transition function, $\delta : (Q \setminus F) \times U \overset{\circ}{\to} 2^{Q \times (U \setminus \{B\}) \times \{R,L\}}$. The variant of a Turing machine that we use in this paper can be described intuitively as follows: it has a semi-infinite tape (bounded to the left) divided into cells (each cell may store exactly one symbol from U). The machine has a central unit storing a state from a finite set of states, and a reading/writing tape head which scans the tape cells; the head cannot write blank symbols. The input is a word over V stored on the tape starting with the leftmost cell while all the other tape cells initially contain the symbol B. When M starts a computation, the tape head scans the leftmost cell and the central unit is in the state q_0. The machine performs moves that depend on the content of the cell currently scanned by the tape head and the current state stored in the central unit. A move consists of: change the state, write a symbol from U on the current cell and move the tape head one cell either to the left (provided that the cell scanned was not the leftmost one) or to the right. An input word is accepted iff after a finite number of moves the Turing machine enters a final state. The machine halts if it reaches a state q and reads a symbol a such that $\delta(q, a)$ is not defined. An instantaneous description (ID for short) of a Turing machine M as above is a word over

$(U \setminus \{B\})^* Q (U \setminus \{B\})^*$. Given an ID $\alpha q \beta$, this means that the tape contents is $\alpha\beta$ followed by an infinite number of cells containing the blank symbol B, the current state is q, and the symbol currently scanned by the tape head is the first symbol of β provided that $\beta \neq \lambda$, or B, otherwise. Usually, we say that an ID $\alpha q \beta$ is final when $q \in F$, and we say that such an ID is blocking when $\delta(q, x)$ is not defined, where x is the first symbol of β (if $\beta \neq \lambda$) or $x = B$ (if $\beta = \lambda$).

A *splicing rule* over a finite alphabet V is a word of the form $u_1 \# u_2 \$ v_1 \# v_2$ such that u_1, u_2, v_1, and v_2 are in V^* and such that \$ and # are two symbols not in V.

For a splicing rule $r = u_1 \# u_2 \$ v_1 \# v_2$ and for $x, y, w, z \in V^*$, we say that r produces (w, z) from (x, y) (denoted by $(x, y) \vdash_r (w, z)$) if there exist some $x_1, x_2, y_1, y_2 \in V^*$ such that $x = x_1 u_1 u_2 x_2$, $y = y_1 v_1 v_2 y_2$, $z = x_1 u_1 v_2 y_2$, and $w = y_1 v_1 u_2 x_2$.

For a language L over V and a set of splicing rules R we define

$$\sigma_R(L) = \{z, w \in V^* \mid (\exists u, v \in L, r \in R)[(u, v) \vdash_r (z, w)]\}.$$

For two disjoint subsets P and F of V and a word x over V, we define the predicates:

$$\phi^s(x; P, F) \equiv P \subseteq alph(x) \wedge F \cap alph(x) = \emptyset$$
$$\phi^w(x; P, F) \equiv P \cap alph(x) \neq \emptyset \wedge F \cap alph(x) = \emptyset.$$

Here, P is the set of *permitting symbols* and F the set of *forbidding symbols*. The first condition ($s = $ strong) requires all permitting symbols and no forbidding symbol to be present in x, whereas for the second ($w = $ weak) at least one permitting and no forbidding symbol should be present in x. For a language $L \subseteq V^*$ and $\beta \in \{w, s\}$,

$$\phi^\beta(L, P, F) = \{x \in L \mid \phi^\beta(x; P, F)\}.$$

If we want to permit all strings of V^*, that is no filtering takes place, we should say $P = V$ for $\beta = w$ or $P = \emptyset$ for $\beta = s$. For simplicity, in those cases we simply write $P = \emptyset$, not specifying β.
A *splicing processor* over V is a 6-tuple (S, A, PI, FI, PO, FO) with

- S a finite set of splicing rules over V,
- A a finite set of auxiliary words over V,
- $PI, FI \subseteq V$ the *input* permitting/forbidding symbols,
- $PO, FO \subseteq V$ the *output* permitting/forbidding symbols.

An *accepting network of splicing processors* (ANSP) is a construct
$\Gamma = (V, U, \langle, \rangle, G, \mathcal{N}, \alpha, x_I, x_O)$, where

- U is the network alphabet and $V \subseteq U$ is the input alphabet.
- $\langle, \rangle \in U - V$ are two special symbols.
- $G = (X_G, E_G)$ is an undirected graph with nodes X_G and edges E_G. We assume G to be complete and without loops.
- $\mathcal{N}$ is a mapping which associates with each node $x \in X_G$ the splicing processor over U, $\mathcal{N}(x) = (S_x, A_x, PI_x, FI_x, PO_x, FO_x)$.
- $\alpha : X_G \to \{s, w\}$ defines the type of the input/output filters, where for each node $x \in X_G$ the input filter ρ and output filter τ are defined as:

$$\rho_x(\cdot) = \phi^{\alpha(x)}(\cdot; PI_x, FI_x),$$
$$\tau_x(\cdot) = \phi^{\alpha(x)}(\cdot; PO_x, FO_x).$$

- $x_I, x_O \in X_G$ are the *input* and *output* nodes, respectively.

The *size* of Γ corresponds to the number of nodes in the graph, i.e. $|X_G|$. A *configuration* of an ANSP Γ is a mapping $C : X_G \to 2^{U^*}$ which associates a set of words to every node of the graph. A configuration can be seen as the sets of words which are present in any node at a given moment. For a word $w \in V^*$ the initial configuration of Γ on w is defined by $C_0^{(w)}(x_I) = \{\langle w \rangle\}$ and $C_0^{(w)}(x) = \emptyset$ for all other $x \in X_G$. By convention, the auxiliary words do not appear in any configuration.

There are two ways to change a configuration: by a splicing step or by a communication step. When changing by a splicing step, each component $C(x)$ of the configuration C is changed according to the set of splicing rules S_x, whereby the words in the set A_x are available for splicing. Formally, configuration C' is obtained in one splicing step from the configuration C, written as $C \Rightarrow C'$, iff for all $x \in X_G$

$$C'(x) = \sigma_{S_x}(C(x) \cup A_x).$$

In a communication step, each processor sends out all strings that can pass the output filter. They are received by all other nodes in the graph, provided they pass the input filter. Note that, according to this definition, strings that can leave a node are sent out even if they cannot pass any input filter. In this case we will say that they are lost. Formally, C' is obtained from C (we write $C' \models C$) iff for all $x \in X_G$

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))).$$

For an ANSP Γ, the computation on an input word w is a sequence of configurations $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \ldots$, where $C_0^{(w)}$ is the initial configuration of Γ on w, $C_{2i}^{(w)} \Rightarrow C_{2i+1}^{(w)}$ and $C_{2i+1}^{(w)} \models C_{2i+2}^{(w)}$, for all $i \geq 0$. A computation *halts* if one of the following two conditions holds:

(1) There exist a configuration in which the set of words existing in the output node x_O is non-empty. This is an *accepting computation*.
(2) There exist two consecutive identical configurations.

The language accepted by Γ is defined as

$$L(\Gamma) = \{w \in V^* \mid \Gamma's \text{ computation on } w \text{ is an accepting computation}\}.$$

We say Γ *decides* a language L if $L(\Gamma) = L$ and Γ halts on every computation.

The reader is referred to [5,6] for the classical time and space complexity classes defined on the standard computing model of Turing machine.

In a similar way, we define two computational complexity measures using ANSP as the computing model. To this aim we consider an ANSP Γ with the input alphabet V that halts on every input. The *time complexity* of the finite computation $C_0^{(x)}$, $C_1^{(x)}, C_2^{(x)}, \ldots, C_m^{(x)}$ of Γ on $x \in V^*$ is denoted by $Time_\Gamma(x)$ and equals m. The *length complexity* of the above computation is defined by $Length_\Gamma(x) = \max\{|w| : w \in C_i^{(x)}(z), 1 \leq i \leq m, z \in X_G\}$.

The time complexity of Γ is the partial function from **N** to **N**,

$$Time_\Gamma(n) = \max\{Time_\Gamma(x) \mid x \in V^*, |x| = n\}.$$

An ANSP Γ is said to be working in polynomial time if $Time_\Gamma(n)$ is bounded by a polynomial.

Analogously, the length complexity of Γ is the partial function from **N** to **N**,

$$Length_\Gamma(n) = \max\{Length_\Gamma(x) \mid x \in V^*, |x| = n\}.$$

As in the former case, an ANSP Γ is said to be working in polynomial length if $Length_\Gamma(n)$ is bounded by a polynomial.

3. ANSPs of size two are computationally complete

In this section, we provide a completeness proof based on the simulation of a deterministic Turing machine.

Theorem 1. *For any language $\mathcal{L}$, accepted (decided) by a deterministic Turing machine M, there exists an* ANSP Γ*, of size* 2*, accepting (deciding) $\mathcal{L}$. Consequently, all recursively enumerable (recursive) languages are accepted (decided) by* ANSPs *of size* 2.

Proof. Let $M = (Q, V, U, q_0, \{q_f\}, \delta, B)$ be a deterministic Turing machine accepting $\mathcal{L}$, with Q the set of states, V and U respectively the input and tape alphabet, q_0 the initial state, B the blank symbol and $\delta : Q \times U \to Q \times U \times \{L, R\}$ the transition function. We assume without loss of generality that M has a single accepting state q_f. Given that M accepts (decides) $\mathcal{L}$, we construct an ANSP $\Gamma = (V, U_\Gamma, \langle', \rangle', K_2, \mathcal{N}, \alpha, 1, 2)$ accepting (deciding) $\mathcal{L}$.

First we define the working alphabet of Γ: $U_\Gamma = U \cup Q \cup \{\%, L, R, \langle, \langle', \rangle'\}$.

Further we define the nodes of the network. The input node 1 is defined by the following:

- $S_1 = \{\langle'\#a\$\langle q_0\#\% \mid a \in U \cup \{\rangle'\}\}$
  (initialization)
  $\cup\{x\#q_1ayz\$L\#bq_2R, xbq_2\#R\$Lq_1a\#yz \mid x \in U \cup \{\langle'\}, y \in U, z \in U \cup \{\rangle'\}, (q_1, a) \to (q_2, b, R) \in \delta, a, b \in U \setminus \{B\}\}$
  $\cup\{x\#q_1\rangle'\$\%\#bq_2\rangle' \mid x \in U \cup \{\langle\}, (q_1, B) \to (q_2, b, R) \in \delta, b \in U \setminus \{B\}\}$
  (right moves)
  $\cup\{x\#cq_1ay\$L\#q_2cbR, xq_2cb\#R\$Lcq_1a\#y \mid x \in U \cup \{\langle\}, y \in U, a, b, c \in U, (q_1, a) \to (q_2, b, L) \in \delta\}$
  $\cup\{x\#cq_1\rangle' \$\%\#q_2cb\rangle' \mid x \in U \cup \{\langle\}, b, c \in U, (q_1, B) \to (q_2, b, L) \in \delta\}$
  (left moves)
- $A_1 = \{\langle q_0\%, \%B\rangle'\} \cup \{LbqR \mid q \in Q, b \in U\} \cup \{LqbcR \mid q \in Q, b, c \in U\} \cup \{\%aq\rangle' \mid a \in U, q \in Q\} \cup \{\%qcb\rangle', \%bq\rangle \mid b, c \in U, q \in Q\}$
- $PO_1 = \{q_f\}$
- $FO_1 = \{L, R, \%\}$

Since $|PO_1| = 1$, we observe that it makes no difference if we use strong or weak filters; indeed, in both cases we have to check if the symbol q_f appears in the communicated words and no symbol from FO_1 appears in these words.

The node 2 has: $S_2 = A_2 = PI_2 = FI_2 = PO_2 = FO_2 = \emptyset$.

The network Γ works on the input $\langle' w\rangle'$ as follows:

- *The initialization phase:* The string is transformed in one splicing step into $\langle q_0 w\rangle'$. Note that all the other strings obtained in this phase have no importance for the rest of the computation. This string cannot pass the output filters so it stays in node 1. We follow the evolution of this string, while being processed by the network.

- *The simulation phase:* At the beginning of this phase the string has the form $\langle w_1q_1aw_2\rangle'$, with $w_1, w_2 \in (U \setminus \{B\})^*$, $a \in U \cup \{\lambda\}$ and $q_1 \in Q$; also, we take $a = \lambda$ only if $w_2 = \lambda$. Note that the string we follow has this form after *the initialization phase*. There are several ways in which the computation may continue:
  1. If $a \neq \lambda$ and $(q_1, a) \to (q_2, b, R) \in \delta$ the string is transformed into the two strings $\langle w_1bq_2R$ and $Lq_1aw_2\rangle'$, after splicing with the axiom Lbq_2R using a rule $x\#q_1ayz\$L\#bq_2R$. Neither of these strings can pass the output filter, so they will remain in node 1. Then, the two strings are spliced by a rule $xbq_2\#R\$Lq_1a\#yz$ and we obtain the string $\langle w_1bq_2w_2\rangle'$, from which a new simulation step can start. The other string Lq_1aR that is produced in these phase does not interfere in the rest of the computation; no other splicing operations can be performed in this phase.
  2. If $a = w_2 = \lambda$ and $(q_1, B) \to (q_2, b, R) \in \delta$ the string is transformed directly into the string $\langle w_1bq_2\rangle'$, after splicing with the axiom $\%bq_2\rangle'$ using rule $x\#q_1\rangle'\$\%\#bq_2\rangle'$. The other string produced, $\%q_i\rangle'$, cannot be used in the rest of the computation; no other splicing operations can be performed in this phase.
  3. If $a \neq \lambda$, $w_1 = w_1'c$ and $(q_1, a) \to (q_2, b, L) \in \delta$ the string is transformed into the two strings $\langle w_1'q_2cbR$ and $Lcq_1aw_2\rangle'$, after splicing with the axiom Lq_2cbR using a rule $x\#cq_1ay\$L\#q_2cbR$. Neither of these strings can pass the output filter, so they will remain in node 1. Then, the two strings are spliced by a rule $xq_2cb\#R\$Lcq_1a\#y$ and we obtain the string $\langle w_1'q_2cbw_2\rangle'$. The other string that are produced, Lcq_1aR cannot be used in the rest of the computation; no other splicing operations can be performed in this phase.
  4. If $a = w_2 = \lambda$, $w_1 = w_1'c$ and $(q_1, B) \to (q_2, b, L) \in \delta$ the string is transformed into the string $\langle w_1'q_2cb\rangle'$, after splicing with the axiom $\%q_2cb\rangle'$ using rule $x\#cq_1\rangle'\$\%\#q_2cb\rangle'$. The other string $\%cq_1\rangle'$ has no influence on the computation, since it is already in A_1; no other splicing operations can be performed in this phase.

  We stress that since M is deterministic at each step there is exactly one possibility to continue the computation in Γ. Moreover, a string $\langle w_1q_1w_2\rangle'$ is transformed in *the simulation phase* in the string $\langle w_3q_2w_4\rangle'$ iff the ID $w_1q_1w_2'$ is transformed into the ID $w_3q_2w_4'$ in the Turing machine M, where w_2' and w_4' are obtained from w_2 and w_4, respectively, by deleting the blank symbols appearing at the right end of these strings.
- *The acceptance phase:* If a string $\langle w_1fw_2\rangle'$, with $f \in F$ and $w_1, w_2 \in U^*$, is obtained after *the simulation phase*, it exits node 1 and enters 2. Then Γ accepts.

No other derivations than the ones mentioned above are possible, thus no other strings than the strings from $L(M)$ can be accepted. In conclusion, it is clear that Γ accepts (decides) exactly the same language as M does, i.e., $L(\Gamma) = L(M)$. Also, it is clear that each step of the Turing machine M is simulated in a constant number of steps by the ANSP Γ. Therefore, if M accepts/rejects the string w in $f(|w|)$ steps, then Γ accepts/rejects w in $\mathcal{O}(f(|w|))$ steps. Finally, since the strings that are obtained during the computation of Γ on a string w are basically configurations of the obtained in the computation of M on w (with at most two new symbols added) it follows that if M accepts / rejects the string w using at most $f(|w|)$ cells of its tape, then $Length_\Gamma(w) \in \mathcal{O}(f(|w|))$. □

We stress the fact that the construction of Γ does not take full advantage of the parallelism of the ANSPs. Though, it is a usual idea to use sequential computations in simulation of deterministic Turing machines (e.g. [14]).

As a final remark of this section, note that Theorem 1 completely solves the problem of designing size-efficient ANSPs deciding (accepting) recursive (recursively enumerable) languages: by definition at least 2 nodes are needed in any ANSP, an input node and an output node.

4. Reduced ANSPs and normal forms

The ANSP we constructed in the previous section uses only output filters. Since ANSPs are proved to be complete (i.e. any ANSP may be simulated by a Turing machine, and vice versa), we obtain the following normal form result as an immediate corollary.

Corollary 2. *For each* ANSP Γ *there exists an equivalent* ANSP Γ_0 *such that for each node x of* Γ_0, $PI_x = FI_x = \emptyset$.

This normal form for ANSPs raises the question whether there are other ways to simplify the model. In other words, can we find reduced types of ANSPs which are still universal? As a partial answer to this question, we show that ANSPs without output filters are also complete. In this case, however, we need an extra node in our system.

Theorem 3. *For each* ANSP Γ *there exists an equivalent* ANSP Γ_0 *of size* 3 *such that for each node x of* Γ_0, $PO_x = FO_x = \emptyset$.

Proof. We show that for every recursively enumerable (recursive) language $\mathcal{L}$ we can construct an ANSP of size three without output filters accepting (deciding) $\mathcal{L}$. Let $M = (Q, V, U, q_0, \{q_f\}, \delta, B)$ be a deterministic Turing machine accepting $\mathcal{L}$, with Q the set of states, V and U respectively the input and tape alphabet, q_0 the initial state, B the blank symbol and $\delta : Q \times U \to Q \times U \times \{L, R\}$ the transition function. We assume without loss of generality that M is deterministic and has a single accepting state q_f.

We construct the ANSP $\Gamma = (V, U_\Gamma, \langle, \rangle, G, \mathcal{N}, \alpha, 3, 1)$, where G is the complete graph with 3 nodes, $U_\Gamma = U \cup \{L, R, Z, \langle, \rangle\}$ and (the value of α is omitted where irrelevant):

- $S_1 = \emptyset$
  $A_1 = \emptyset$
  $PI_1 = \{q_f\}$
  $FI_1 = \{L, R\}$
  $PO_1 = FO_1 = \emptyset$

In what follows $a, b, c, d \in U \cup \{\langle, \rangle\}$, and $q_i, q_j \in Q$.

- $S_2 = \{$
  $c\#q_i\rangle\$L\#q_iB\rangle R$ (right end of the tape)
  $c\#q_ia\$L\#bq_jR$ for $(q_i, a) \to (q_j, b, R) \in \delta$ (right move)
  $\lambda\#cq_ia\$L\#q_jcbR$ for $(q_i, a) \to (q_j, b, L) \in \delta\}$ (left move)
  $A_2 = \{Lbq_jR \mid (q_i, a) \to (q_j, b, R) \in \delta\} \cup \{Lq_jcbR \mid (q_i, a) \to (q_j, b, L) \in \delta\}$
  $\cup\{Lq_iB\rangle R\}$
  $FI_2 = \{L, R\}$
  $PI_2 = PO_2 = FO_2 = \emptyset$
- $S_3 = \{$
  $\langle a\#\lambda\$\langle q_0a\#Z$ for $a \in V$ (initialization)
  $cq_iB\rangle\#R\$Lq_i\rangle\#\lambda$ (right end of the tape)
  $cbq_j\#R\$Lq_ia\#\lambda$ for $(q_i, a) \to (q_j, b, R) \in \delta$ (right move)
  $dq_jcb\#R\$Lcq_ia\#\lambda$ for $(q_i, a) \to (q_j, b, L) \in \delta\}$ (left move)
  $A_3 = \{\langle q_0aZ \mid a \in V\}$
  $PI_3 = \{L, R\}$
  $FI_3 = PO_3 = FO_3 = \emptyset$
  $\alpha(3) = w$

The idea behind this systems is basically the same as the proof of Theorem 1, but in this case the simulation is done by moving back and and forth between nodes 2 and 3. Node 3 is the input node, where the input $\langle w\rangle$ is converted into $\langle q_0w\rangle$, which is sent out in the communication step. It will reach only node 2 (unless q_0 is the final state). Now, the moves of M are simulated. When a final state is reached, that is we get to a word of the form $\langle w_1q_fw_2\rangle$, with $w_1, w_2 \in U^*$, this word can pass the input filter of node 1, and input w is accepted.

In node 2, we can start simulating the moves of M. For a right move $(q_i, a) \to (q_j, b, R)$ on a configuration $\langle wq_iaw'\rangle$ we apply rule $c\#q_ia\$L\#bq_jR$ using Lbq_jR from A_2, giving $\langle wbq_jR$ and $Lq_iaw'\rangle$, which are passed to node 3. In node 3, the two words are combined by rule $cbq_j\#R\$Lq_ia\#\lambda$ to give $\langle wbq_jw'\rangle$, the new configuration. This word is sent back to node 2, where the simulation of the next move begins. Left moves are simulated in a similar way, using a rule $\lambda\#cq_ia\$L\#q_jcbR$ in node 2 and a rule $dq_jcb\#R\$Lcq_ia\#\lambda$ in node 3. If M needs more tape space on the right, we can insert a blank symbol B at the end of our configuration using rules $c\#q_i\rangle\$L\#q_iB\rangle R$ in node 2 and $cq_iB\rangle\#R\$Lq_i\rangle\#\lambda$ in node 3. Indeed, when we have a configuration of the form $\langle wq_i\rangle$, rule $c\#q_i\rangle\$L\#q_iB\rangle R$ yields $\langle wq_iB\rangle R$ and $Lq_i\rangle$, which combine in node 3 to give $Lq_i\rangle R$ and $\langle wq_iB\rangle$. This last string is sent out to node 2, where the simulation resumes.

It should be clear from this explanation that Γ accepts the input string $\langle w\rangle$ if M reaches a final state on input w; moreover, Γ halts on the input $\langle w\rangle$ if M halts on the input string w. To see that Γ accepts exactly the same words M does (and halts on the same words as M), recall that since M is deterministic there is only one possible move at every step, thus at all times only one word representing a configuration is present in node 2 or two parts of this configuration in node 3. Moreover, all other strings do not produce words that can interfere in the derivation. The words LvR, $v \in U \cup Q \cup \{\rangle\}$ produced in node 3 are sent out and cannot pass any input filter. In node 2, only strings involved in the derivation are produced and sent to node 3.

Thus, Γ accepts on input $\langle w\rangle$ if and only if M reaches a final state on input w. Also, Γ halts on the input $\langle w\rangle$ if and only if M halts on the input string w. Moreover, Γ uses only input filters. This proves the theorem. □

Remark 4. We remark here that the size bound of 3 nodes is strict, since without output filters no strings can be retained in their current component. This means that at least two non-output nodes are needed for non-trivial computation.

5. Complexity results

In [10] it is shown that ANSPs of size 7 can accept languages in **NP** in polynomial time. Since in our construction we simulated a deterministic Turing machine, no such claim about ANSPs of two nodes can be derived from Theorem 1. However, in this section we show that all **NP**-problems can be decided in polynomial time by ANSPs of size 3.

First, we present a more involved construction of an ANSP with 3 nodes that can simulate in parallel the computations of non-deterministic Turing machines. We will use a variation of the rotate-and-simulate technique introduced in [20] and the 'counting-down' mechanism often used in proofs for test-tube systems and time-varying H systems, e.g. [4,22,15].

Lemma 5. *For any language L, accepted (decided) by a non-deterministic Turing Machine M, there exists an* ANSP Γ*, of size* 3*, accepting (deciding) L.*

Proof. Let $M = (Q, V, U, q_0, \{q_f\}, \delta, B)$ be a non-deterministic Turing machine, with Q the set of states, V and U respectively the input and tape alphabet, q_0 the initial state, B the blank symbol and $\delta : Q \times U \to Q \times U \times \{L, R\}$ the transition function. We assume without loss of generality that M has a single accepting state q_f.

Let $|U \cup V| = n$ and $K = \{\langle^i, \rangle^i, \langle^{i'}, \rangle^{i'} \mid 0 \le i \le 2n\}$. We assume some ordering such that $U \cup V = \{k_1, \ldots, k_n\}$ and each k_i identifies a unique element of $U \cup V$. We construct the ANSP $\Gamma = (V, U_\Gamma, \langle, \rangle', G, \mathcal{N}, \alpha, 3, 1)$, where G is the complete graph with 3 nodes, $U_\Gamma = U \cup K \cup \{Z, E, \langle, \rangle, \langle', \rangle'\}$, all $\alpha(i) = w$, $1 \le i \le 3$ and:

- $S_1 = \emptyset$
  $A_1 = \emptyset$
  $PI_1 = \{q_f\}$
  $FI_1 = \{Z\} \cup \{\langle^i, \langle^{i'}, \rangle^i, \rangle^{i'} \mid 0 \le i \le 2n\}$
  $PO_1 = FO_1 = \emptyset$

In what follows $a, b, c, d \in U \cup \{\langle, \rangle\}$, $m \in U \cup Q$ and $q, q_i, q_j \in Q$.

- $S_2 = \{$
  (simulate)
  $\langle q_i a \# c \$ \langle' b q_j \# Z$ for $(q_i, a) \to (q_j, b, R) \in \delta$
  $\langle^{0'} q_i a \# c \$ \langle' b q_j \# Z$ for $(q_i, a) \to (q_j, b, R) \in \delta$
  $\langle c q_i a \# d \$ \langle' c b \# Z$ for $(q_i, a) \to (q_j, b, L) \in \delta$
  $\langle^{0'} c q_i a \# d \$ \langle' c b \# Z$ for $(q_i, a) \to (q_j, b, L) \in \delta$
  (right end of tape)
  $c \# q E \rangle \$ Z \# q B E \rangle$
  (start rotate)
  $\langle q \# c \$ \langle^i k_i q \# Z$ for $1 \le i \le n$
  $c \# k_i \rangle \$ Z \#)^i$ for $1 \le i \le n$
  $\langle k_{i-n} \# q c \$ \langle^i \# Z$ for $n+1 \le i \le 2n$
  $c \# \rangle \$ Z \# k_{i-n} \rangle^i$ for $n+1 \le i \le 2n$
  (decrease counter)
  $\langle^{i'} \# m \$ \langle^{i-1} \# Z$ for $1 \le i \le 2n$
  $a \#)^{i'} \$ Z \#)^{i-1}$ for $1 \le i \le 2n\}$
  $A_2 = \{\langle' b q_j Z \mid q_i, a) \to (q_j, b, R) \in \delta\} \cup \{\langle' c b Z \mid q_i, a) \to (q_j, b, L) \in \delta\}$
  $\cup \{\langle^i k_i q Z, Z\rangle^i \mid 1 \le i \le n\} \cup \{\langle^i Z, Z k_i\rangle^i \mid n+1 \le i \le 2n\} \cup \{Z q B E\rangle \mid q \in Q\} \cup \{\langle^i Z, Z\rangle^i \mid 0 \le i \le 2n-1\}$
  $PI_2 = \{\langle, \langle^{0'}\} \cup \{\}^{i'} \mid 1 \le i \le 2n\}$
  $FI_2 = \{Z\}$
  $PO_2 = \{\}\} \cup \{\langle^i \mid 0 \le i \le 2n\}$
  $FO_2 = \emptyset$
- $S_3 = \{$
  (initialize)
  $\langle \# a b \$ \langle q_0 \# Z$
  $\lambda \# \rangle' \$ Z \# E \rangle$
  (return simulate)
  $\langle' \# m \$ \langle \# Z$
  (prime counter)
  $\langle^i \# m \$ \langle^{i'} \# Z$ for $0 \le i \le 2n$
  $m \#\rangle^i \$ Z \#\rangle^{i'}$ for $1 \le i \le 2n$
  (resume computation)
  $m \#\rangle^0 \$ Z \#\rangle\}$
  $A_3 = \{\langle Z, \langle q_0 Z, Z E\rangle, Z\rangle\} \cup \{Z\rangle^{i'} \mid 1 \le i \le 2n\}$
  $\cup \{\langle^{i'} Z \mid 0 \le i \le 2n\}$
  $PI_3 = \{\langle'\} \cup \{\}^i \mid 0 \le i \le 2n\}$
  $FI_3 = \{Z\}$
  $PO_3 = \{\}\} \cup \{\langle^{i'} \mid 1 \le i \le 2n\}$
  $FO_3 = \emptyset$

We show that $\langle w\rangle'$ is accepted by Γ if and only if w is accepted by M.[1] We start with the 'if'-part. The computation of Γ begins with the initialization phase. The word $\langle w\rangle'$ is converted to $\langle q_0 w\rangle'$ by the rule $\langle \# a b \$ \langle q_0 \# Z$. This word cannot pass

[1] Note that we use $\rangle'$ for the input. This is turned into $\rangle$ in the initialization phase. We prefer the unusual input in order to avoid a proliferation of primes in the proof.

PO_3 so it stays in node 3. Next, $\lambda\#\rangle'\$Z\#E\rangle$ is applied to give $\langle q_0 wE\rangle$. The symbol E denotes the end of the tape of M. This string is passed to node 2. This concludes the initialization phase.

In node 2, we have two possibilities, simulating a move of M or rotating a symbol. We can simulate a right move $(q_i, a) \to (q_j, b, R)$ by applying the rule $\langle q_i a\#c\$\langle' bq_j\#Z$. Starting from a string $\langle q_i aw\rangle$ this yields $\langle' bq_j w\rangle$. This string is sent out to node 3, where it is converted to $\langle bq_j w\rangle$ by $\langle'\#m\$\langle\#Z$, which is passed back to node 2. Thus we have the desired result of the move. If we have a word $\langle cq_i aw\rangle$ and $(q_i, a) \to (q_j, b, L)$ is a valid move in M, we obtain the desired result $\langle q_j cbw\rangle$ in node 2 by applying $\langle cq_i a\#d\$\langle' cb\#Z$ and, in node 3, $\langle'\#m\$\langle\#Z$.

Note that we simulate the moves of M only at the left end of the string. This means that if we have a sequence of two left moves or two right moves, we need to rotate symbols to make the simulation possible. For instance, from a string $\langle dqaw\rangle$ we need to go to $\langle qawd\rangle$ to allow for the simulation of a move $(q, a) \to (q_j, b, R)$. This rotating is done as follows. Suppose $d = k_i$. Then in node 2, we can apply a rule $c\#k_i\rangle\$Z\#\rangle^i$ to convert $\langle dqaw\rangle$ to $\langle dqawd\rangle^i$. This string does not pass the output filter, so it stays in node 2. In the next step, $\langle dqawd\rangle^i$ becomes $\langle^i qawd\rangle^i$ by a rule $\langle q\#c\$\langle^i k_i q\#Z$. Now the counter is progressively lowered by moving the string back and forth between nodes 2 and 3. The word $\langle^i qawd\rangle^i$ passes to node 3, where a rule $m\#\rangle^i\$Z\#\rangle i'$ gives $\langle^i qawd\rangle^{i'}$ (remains in node 3) and $\langle^{i'}\#m\$\langle^{i'}\#Z$ yields $\langle^{i'} qawd\rangle^{i'}$ (to node 2). In node 2, rules $a\#\rangle^{i'}\$Z\#\rangle^{i-1}$ (result remains in 2) and $\langle^{i'}\#m\$\langle^{i-1}\#Z$ give $\langle^{i-1} qawd\rangle^{i-1}$, which passes to node 3. This process is repeated until arriving at $\langle^0 qawd\rangle^0$. This string arrives in node 3, where $\langle^i\#m\$\langle^{i'}\#Z$ (result remains) and $m\#\rangle^0\$Z\#\rangle$ convert the string to $\langle^{0'} qawd\rangle$. This string is passed to node 2, where a simulation rule $\langle^{0'} q_i a\#c\$\langle' bq_j\#Z$ could be applied. Rotating a symbol to the left is done similarly. To avoid interference between right and left rotation of the same symbol, for the left rotation of symbol k_i we use counter $i + n$. Note that the rotated strings begin with the symbol $\langle^{0'}$. For strings starting with $\langle^{0'}$, S_2 contains the same simulation rules as for $\langle$, but no rotation rules. This avoids fruitless rotations, since we know that the head of a Turing machine moves at most one symbol per move.

The special symbol E cannot be rotated, which enforces that the head of M never goes beyond the left end of the tape. If M needs more tape cells on the right, a rule $c\#qE\rangle\$Z\#qBE\rangle$ can be applied to insert a blank symbol between the head and the end-of-tape symbol. The result of this does not leave node 2, so simulation can continue as described above. All necessary auxiliary strings, all containing the symbol Z, for the described process are present in the sets A_2 and A_3. Finally, if M reaches a final state, the simulation in Γ will yield a word of the form $\langle wq_f w'\rangle$ in node 3. This word passes the output filters and the input filters of node 1, causing Γ to accept.

For the 'only if'-part of the proof, we show that only by the way described above we can get an accepting computation of Γ. First of all, in the initialization phase we could apply the rule $\lambda\#\rangle'\$Z\#E\rangle$ first. The result $\langle wE\rangle$ can pass to node 2. However, since there is no state symbol, no simulation or acceptance is possible. We can only rotate, but after arriving at a string $\langle^{0'} v\rangle$ no further rule applications are possible. For the simulation steps, they can clearly only give the described result. Also in node 3, there is only one applicable rule. The old configuration can leave node 2, but not enter any other node.

For the rotation, the decreasing procedure involves two splicing steps in the same node before being passed to the other node. If we apply these two rules in the inverse order with respect to our description above, it is easy to verify that we get a string that can leave the node, but not enter the other node. Now, when starting the rotation phase, it can be that the new symbol we attach to one side does not correspond to the symbol removed on the other side. We show that these incorrect rotations do not lead to accepting computations. Suppose we have $\langle qawk_j\rangle$ and we apply $\langle q\#c\$\langle^i k_i q\#Z$ for some $i \neq j$. This means we get $\langle^i k_i qaw\rangle^j$ for $i \neq j$ in the next step. For rotation to the right we can get $\langle^i wk_j\rangle^j$ for $i \neq j$. Then we can have two cases:

- $i < j$ If $i < j$, by the decreasing procedure, we arrive at a string $\langle^0 v\rangle^l$ in node 2, with $l > 0$. This string is passed to node 3. There we can apply $m\#\rangle^i\$Z\#\rangle i'$ to give $\langle^0 v\rangle^{l'}$. This string cannot pass the output filter, so it stays in node 3. Then, the only applicable rule is $\langle^i\#m\$\langle^{i'}\#Z$. This gives $\langle^{0'} v\rangle^{l'}$. This string cannot leave node 3 and no more rules can be applied to it, so it is 'trapped' and cannot lead to acceptance. If the two rules are applied in the inverse order, we first get $\langle^{0'} v\rangle^l$, which cannot pass the output filter, and then the same string $\langle^{0'} v\rangle^{l'}$.
- $i > j$ If $i > j$, we arrive at a string $\langle^l v\rangle^0$, $l > 0$ in node 2. This string passes to node 3, where two rules can apply. If we apply a rule $\langle^i\#m\$\langle^{i'}\#Z$, we get $\langle^{l'} v\rangle^0$. Applying $m\#\rangle^0\$Z\#\rangle$ gives $\langle^l v\rangle$. Both these strings can pass the output filter, but no input filter, so they are lost.

This shows that only correct simulations of M can lead to acceptance, and thus $L(\Gamma) = L(M)$. Moreover, whenever there is more than one move available to M on a given computation, Γ will simulate all moves, using the multiplicity inherent in the model. As a final remark, from the above explanations follows immediately that Γ halts on an input string $\langle w\rangle'$ if and only if M stops on the input string w. □

From the previous theorem we obtain the following result:

Theorem 6. *All languages in* ***NP*** *can be decided by ANSPs of size* 3 *working in polynomial time.*

Proof. If a language L is in **NP**, there exists a non-deterministic Turing machine M accepting all $w \in L$ in time $f(|w|)$, where $f(n)$ is a polynomial function. Moreover, there exists a $f(n)$-bounded Turing machine M' which simulates M but halts after $f(n)$ steps. By Lemma 5, this means that there exists an ANSP Γ deciding L. Moreover, for each move of M', Γ needs 4 steps for simulation, at most 1 step for expanding the work tape and at most $4 \cdot 2 \cdot m$ rotation steps, where $m = |U|$ and V and U

are the input and tape alphabet of M. Thus, $Time_\Gamma(w) \leq 40m \cdot f(|w|) + 1$ for all w (the term $+1$ comes from our assumption in Lemma 5 that M' has a single final state). This is clearly polynomial. □

From the proof of Theorem 1 it follows that:

Theorem 7. *All languages in **PSPACE** can be decided by ANSPs of size 2 working in polynomial length.*

Proof. If a language L is in **PSPACE**, then there exists a deterministic Turing machine M accepting all $w \in L$ in space $f(|w|)$, where $f(n)$ is a polynomial function. From the remarks made at the end of the proof of Theorem 1, this means that there exists an ANSP Γ deciding L working in polynomial space. □

6. On small and fast universal ANSPs

We recall from [10] a straightforward way of encoding an arbitrary complete ANSP using the fixed alphabet:

$$A = \{\$, \#, *, s, w, 0, 1, \spadesuit, \bullet\}.$$

Let $\Gamma = (V, U, \langle, \rangle, K_n, \mathcal{N}, \alpha, x_I, x_O)$ be an ANSP, where
– $V = \{a_1, a_2, \ldots, a_m\}$, and $U = \{a_1, a_2, \ldots, a_p\}$, $p \geq m + 2$; assume that $a_{m+1} = \langle$ and $a_{m+2} = \rangle$.
– the underlying graph of Γ is the complete graph K_n with n nodes $x_1, x_2, \ldots, x_n$, such that $x_1 = x_I$ and $x_2 = x_O$.
We encode every symbol a_i of U, and denote this encoding by $code(a_i)$, in the following way:

$$code(a_i) = \begin{cases} 10^i, & 1 \leq i \leq m \\ \spadesuit 0^i, & m+1 \leq i \leq p. \end{cases}$$

Given z, a word over U as above, we define its encoding $code(z)$ as follows:

$$code(\lambda) = 1, \qquad code(b_1 b_2 \ldots b_k) = code(b_1)code(b_2)\ldots code(b_k), \quad b_i \in U,\ 1 \leq i \leq k.$$

Let $L \subseteq U^*$ be a finite language, $L = \{z_1, \ldots, z_k\}$. We encode this language by the word $code(L) = \bullet code(z_1) \bullet code(z_2) \bullet \cdots \bullet code(z_k) \bullet$. The empty language is encoded by $code(\emptyset) = \bullet$.

Further, the splicing rule $r = x\#y\$u\#v$ is encoded as:

$$code(r) = *code(x) * code(y) * code(u) * code(v) * .$$

A set of splicing rules $R = \{r_1, \ldots, r_m\}$ is encoded:

$$code(R) = \bullet code(r_1) \bullet code(r_2) \bullet \cdots \bullet code(r_m) \bullet .$$

For each node x, $code(\mathcal{N}(x))$ is

$$\#code(S_x)\#code(A_x)\#code(PI_x)\#code(FI_x)\#code(PO_x)\#code(FO_x)\#,$$

and $code(x) = \#code(\mathcal{N}(x))\alpha(x)\#$. We now describe the way Γ is encoded. This is:

$$code(\Gamma) = \$code(K_n)\$code(x_1)\$code(x_2)\$\ldots\$code(x_n)\$,$$

where $code(K_n) = \spadesuit^n$.

In [10] it was proved that:

Theorem 8. *There exists a deterministic Turing machine T_U, with the input alphabet A, satisfying the following conditions on any input $code(\Gamma)code(z)$, where $\Gamma = (V, U, \langle, \rangle, G, \mathcal{N}, \alpha, x_I, x_O)$ is an arbitrary unrestricted ANSP and z is a word over the input alphabet of Γ:*
(i) *T_U halts on the input $code(\Gamma)code(z)$ if and only if Γ halts on the input z.*
(ii) *$code(\Gamma)code(z)$ is accepted by T_U if and only if z is accepted by Γ.*

The Turing machine constructed in the proof of Theorem 8 does not seem to have good computational properties, even in the cases when Γ works efficiently. We recall from [10] that T_U effectively keeps track of all the configurations that are reached during the computation of Γ on z.

Further we make several straightforward comments on the existence of universal ANPSs. Since for each deterministic Turing machine one can construct an ANSP of size 2 accepting (deciding) the same language, from Theorem 8 it results the following result:

Theorem 9. *There exists an ANSP of size 2, Γ_U, with the input alphabet A, satisfying the following conditions on any input $code(\Gamma)code(w)$, where Γ is an arbitrary ANSP and w is a word over the input alphabet of Γ:*

- *Γ_U halts on the input $code(\Gamma)code(w)$ if and only if Γ halts on the input w.*
- *$code(\Gamma)code(w)$ is accepted by Γ_U if and only if w is accepted by Γ.*

Nevertheless, ANSPs are efficient simulators of Turing machines. In [17] are given universal Turing machines that simulate the input Turing machines in quadratic time, i.e. given a Turing machine M that runs on the input w in time t the universal Turing machine runs in time $\mathcal{O}(t^2)$ on the input $code(M)code(w)$ ($code$ being an encoding of Turing machines that allows the existence of fast universal Turing machines). Therefore, we can provide the following result:

Theorem 10. *There exists an ANSP of size 2, Γ_E, satisfying the following conditions on any input $code(M)code(w)$, where M is an arbitrary deterministic Turing machine and w is a word over the input alphabet of M:*

- *Γ_E halts on the input $code(M)code(w)$ if and only if M halts on the input w.*
- *$code(M)code(w)$ is accepted by Γ_U if and only if w is accepted by M.*

Moreover, if M runs on the input w in time t then Γ_E runs in time $\mathcal{O}(t^2)$ on the input $code(M)code(w)$.

The reversal of this theorem does not necessarily hold, i.e. deterministic Turing machine are not necessarily fast simulators of ANSPs. Indeed, in [10] it is shown that all NP-languages can be accepted in polynomial time by ANSPs of constant size. Therefore, if one would build a deterministic Turing machine that efficiently (i.e. in polynomial time) simulates ANSPs, it would follow that all NP-languages are acceptable in Turing deterministic polynomial time. This holds if and only if $P = NP$.

On the other hand, non-deterministic Turing machines can be efficiently simulated by ANSPs with 3 nodes. Indeed, one can easily build an universal non-deterministic Turing machine M_U that efficiently simulates the computation of the non-deterministic Turing machines whose code is given as input on the word whose code is also given as input. This can be done in the following way: this universal Turing machines implements a similar strategy to that implemented by a fast deterministic universal Turing machine (for example one of the universal Turing machines working in quadratic time, from [17]), the only difference being that when it must begin the simulation of a move of the non-deterministic Turing machine given as input, it chooses non-deterministically a possible move and starts simulating it (instead of simulating the only possible move). It is clear that if the Turing machine given as input runs on the input word in non-deterministic time t then the universal machine runs in time $\mathcal{O}(t^2)$ on its input. Based on Lemma 5 we can construct an ANSP Γ_E that simulates the behavior of M_U. Consequently, we state the following result:

Theorem 11. *There exists an ANSP of size 3, Γ_E, satisfying the following conditions on any input $code(M)code(w)$, where M is an arbitrary non-deterministic Turing machine and w is a word over the input alphabet of M:*

- *Γ_E halts on the input $code(M)code(w)$ if and only if M halts on the input w.*
- *$code(M)code(w)$ is accepted by Γ_U if and only if w is accepted by M.*

Moreover, if M runs on the input w in non-deterministic time t then Γ_E runs in time $\mathcal{O}(t^2)$ on the input $code(M)code(w)$.

To prove the converse of the last theorem one would need, for example, a result stating that for each ANSP one can build a corresponding non-deterministic Turing machine that simulates the ANSP efficiently. We leave this as an open problem.

We state also as an open problem that of designing an universal ANSP that simulates efficiently ANSPs.

7. Final remarks

As mentioned in the introduction, acceptance in ANSPs is defined in such a way that at least two nodes are needed to accept any non-trivial language. This means that we completely settled the issue with respect to this descriptional complexity measure: the most simple ANSPs are complete. We leave the question of whether two nodes are enough, or not, to simulate efficiently any non-deterministic Turing machine (thus decide all problems in **NP** in polynomial time) as an open problem; another unsolved problem is to simulate ANSPs efficiently using Turing machines. Finally, it remains open if we can design a small universal ANSP able to simulate efficiently the ANSPs given as input.

Our normal form results for reduced versions of ANSPs also point to other directions of research. Can we find other reduced types of ANSPs which are still universal? In Theorem 3, forbidding filters are only used in one place. Can we find complete systems with only permitting filters? For such reduced systems it would be interesting to investigate which price we have to pay in terms of size of the systems, or in computational resources. Also, it would be interesting to find restrictions that lead to weaker language classes.

Acknowledgement

The first author's work was supported by Research Grant BES-2004-6316 of the Spanish Ministry of Education and Science. The work of the second and the third author was partially supported by the Romanian Ministry of Education and Research (PN-II Program, Project *GlobalComp - Models, semantics, logics and technologies for global computing*).

References

[1] E. Csuhaj-Varjú, L. Kari, G. Păun, Test tube distributed systems based on splicing, Comput. Artif. Intell. 15 (1996) 211–232.
[2] L. Errico, C. Jesshope, Towards a new architecture for symbolic processing, in: Artificial Intelligence and Information-Control Systems of Robots '94, World Scientific, Singapore, 1994, pp. 31–40.

[3] S.E. Fahlman, G.E. Hinton, T.J. Seijnowski, Massively parallel architectures for AI: NETL, THISTLE and Boltzmann machines, in: Proc. AAAI National Conf. on AI, William Kaufman, Los Altos, 1983, pp. 109–113.
[4] R. Freund, E. Csuhaj-Varjú, F. Wachtler, Test tube systems with cutting/recombination operations, in: Pacific Symposium on Biocomputing'97, World Scientific, Singapore, 1997, pp. 163–175.
[5] J. Hartmanis, P.M. Lewis II, R.E. Stearns, Hierarchies of memory limited computations, in: Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logical Design, 1965, pp. 179–190.
[6] J. Hartmanis, R.E. Stearns, On the computational complexity of algorithms, Trans. Amer. Math. Soc. 117 (1965) 533–546.
[7] W.D. Hillis, The Connection Machine, MIT Press, 1985.
[8] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison Wesley, 1979.
[9] F. Manea, C. Martin-Vide, V. Mitrana, Accepting networks of splicing processors: Complexity results, Theoret. Comput. Sci. 371 (1–2) (2007) 72–82; Erratum to: Accepting networks of splicing processors: Complexity results [Theoret. Comput. Sci. 371(2007) 72–82], Theoret. Comput. Sci. 378(1) 131.
[10] F. Manea, C. Martin-Vide, V. Mitrana, All NP-problems can be solved in polynomial time by accepting networks of splicing processors of constant size, in: DNA12, in: Lecture Notes in Computer Science, vol. 4287, 2006, pp. 47–57.
[11] F. Manea, C. Martín-Vide, V. Mitrana, On the size complexity of universal accepting hybrid networks of evolutionary processors, Math. Structures Comput. Sci. 17 (2007) 753–771.
[12] F. Manea, V. Mitrana, All NP-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size, Inf. Process. Lett. 103 (2007) 112–118.
[13] M. Margenstern, V. Mitrana, M. Perez-Jimenez, Accepting hybrid networks of evolutionary systems, in: Lecture Notes in Computer Science, vol. 3384, Springer-Verlag, Berlin, 2005, pp. 235–246.
[14] M. Margenstern, Y. Rogozhin, Time-varying distributed H systems of degree 1 generate all recursively enumerable languages, in: M. Ito, Gh. Paun, S. Yu (Eds.), Words, Semigroups, and Transductions, World Scientific Publishing, Singapore, 2001, pp. 329–340.
[15] M. Margenstern, Y. Rogozhin, S. Verlan, Time-varying distributed H systems of degree 2 can carry out parallel computations, in: DNA8, in: Lecture Notes in Computer Science, vol. 2568, 2003, pp. 326–336.
[16] C. Martín-Vide, V. Mitrana, Networks of evolutionary processors: Results and perspectives, in: Molecular Computational Models: Unconventional Approaches, Idea Group Publishing, Hershey, 2005, pp. 78–114.
[17] T. Neary, D. Woods, Small fast universal Turing machines, Theoret. Comput. Sci. 362 (2006) 171–195.
[18] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
[19] Gh. Păun, Distributed architectures in DNA computing based on splicing: Limiting the size of components, in: Unconventional Models of Computation, Springer-Verlag, Berlin, 1998, pp. 323–335.
[20] Gh. Păun, Regular extended H systems are computationally universal, J. Autom. Lang. Comb. 1 (1) (1996) 27–36.
[21] G. Păun, DNA computing; Distributed splicing systems, in: J. Mycielski, G. Rozenberg, A. Salomaa (Eds.), Structures in Logic and Computer Science. A Selection of Essays in Honor of A. Ehrenfeucht, in: LNCS, vol. 1261, Springer-Verlag, Berlin, 1997, pp. 351–370.
[22] A. Păun, On time-varying H systems, Bull. Eur. Assoc. Theor. Comput. Sci. EATCS 67 (1999) 157–164.
[23] D. Sankoff, G. Leduc, N. Antoine, B. Paquin, B.F. Lang, R. Cedergren, Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome, Proc. Natl. Acad. Sci. USA 89 (1992) 6575–6579.